Chapter 7. The Process

Introduction

A process is an OS abstraction that enables us to look at files and programs as their time image. This chapter discusses processes, the mechanism of creating a process, different states of a process and also the ps command with its different options. A discussion on creating and controlling background jobs will be made next. We also look at three commands viz., at, batch and cron for scheduling jobs. This chapter also looks at nice command for specifying job priority, signals and time command for getting execution time usage statistics of a command.

Objectives

- Process Basics
- ps: Process Status
- Mechanism of Process Creation
- Internal and External Commands
- Process States and Zombies
- Background Jobs
- nice: Assigning execution priority
- Processes and Signals
- job Control
- at and batch: Execute Later
- cron command: Running Jobs Periodically
- time: Timing Usage Statistics at process runtime

1. Process Basics

UNIX is a multiuser and multitasking operating system. *Multiuser* means that several people can use the computer system simultaneously (unlike a single-user operating system, such as MS-DOS). *Multitasking* means that UNIX, like Windows NT, can work on several tasks concurrently; it can begin work on one task and take up another before the first task is finished.
When you execute a program on your UNIX system, the system creates a special environment for that program. This environment contains everything needed for the system to run the program as if no other program were running on the system. Stated in other words, a process is created. A process is a program in execution. A process is said to be born when the program starts execution and remains alive as long as the program is active. After execution is complete, the process is said to die.

The kernel is responsible for the management of the processes. It determines the time and priorities that are allocated to processes so that more than one process can share the CPU resources.

Just as files have attributes, so have processes. These attributes are maintained by the kernel in a data structure known as process table. Two important attributes of a process are:

1. The Process-Id (PID): Each process is uniquely identified by a unique integer called the PID, that is allocated by the kernel when the process is born. The PID can be used to control a process.
2. The Parent PID (PPID): The PID of the parent is available as a process attribute.

There are three types of processes viz.,

1. Interactive: Initiated by a shell and running in the foreground or background
2. batch: Typically a series of processes scheduled for execution at a specified point in time
3. daemon: Typically initiated at boot time to perform operating system functions on demand, such as LPD, NFS, and DNS

The Shell Process

As soon as you log in, a process is set up by the kernel. This process represents the login shell, which can be either sh(Bourne Shell), ksh(korn Shell), bash(Bourne Again Shell) or csh(C Shell).

Parents and Children

When you enter an external command at the prompt, the shell acts as the parent process, which in turn starts the process representing the command entered. Since every parent has a parent, the ultimate ancestry of any process can be traced back to the first process (PID 0) that is set up when the system is booted. It is analogous to the root directory of the file system. A process can have only one parent. However, a process can spawn multiple child processes.

Wait or not Wait?

A parent process can have two approaches for its child:

- It may wait for the child to die so that it can spawn the next process. The death of the child is intimated to the parent by the kernel. Shell is an example of a parent that waits for the child to terminate. However, the shell can be told not to wait for the child to terminate.
- It may not wait for the child to terminate and may continue to spawn other processes. init process is an example of such a parent process.

2. ps: Process Status

Because processes are so important to getting things done, UNIX has several commands that enable you to examine processes and modify their state. The most frequently used command is ps, which prints out the process status for processes running on your system. Each system has a slightly different version of the ps command, but there are two main variants, the System V version (POSIX) and the Berkeley version. The following table shows the options available with ps command.

POSIX	**BSD**	**Significance**
-f	f	Full listing showing PPID of each process
-e or –A	aux	All processes (user and system) processes

-u *user*	U *user*	Processes of user *user only*
-a		Processes of all users excluding processes not associated with terminal
-l	l	Long listing showing memory related information
-t *term*	t *term*	Processes running on the terminal *term*

Examples

```
$ ps
PID  TTY   TIME     CMD
4245 pts/7 00:00:00 bash
5314 pts/7 00:00:00 ps
```

The output shows the header specifying the PID, the terminal (TTY), the cumulative processor time (TIME) that has been consumed since the process was started, and the process name (CMD).

```
$ ps -f
UID    PID   PPID  C STIME    TTY   TIME COMMAND
root   14931 136   0 08:37:48 ttys0 0:00 rlogind
sartin 14932 14931 0 08:37:50 ttys0 0:00 -sh
sartin 15339 14932 7 16:32:29 ttys0 0:00 ps –f
```

The header includes the following information:

UID – Login name of the user
PID – Process ID
PPID – Parent process ID
C – An index of recent processor utilization, used by kernel for scheduling
STIME – Starting time of the process in hours, minutes and seconds
TTY – Terminal ID number
TIME – Cumulative CPU time consumed by the process
CMD – The name of the command being executed

System processes (-e or –A)

Apart from the processes a user generates, a number of system processes keep running all the time. Most of them are not associated with any controlling terminal.
They are spawned during system startup and some of them start when the system goes into multiuser mode. These processes are known as daemons because they are called without a specific request from a user. To list them use,

```
$ ps –e
PID    TTY     TIME  CMD
0      ?       0:34  sched
1      ?       41:55 init
23274  Console 0:03  sh
272    ?       2:47  cron
7015   term/12 20:04 vi
```

3. Mechanism of Process Creation

There are three distinct phases in the creation of a process and uses three important system calls viz., fork, *exec*, and wait. The three phases are discussed below:

- Fork: A process in UNIX is created with the fork system call, which creates a copy of the process that invokes it. The process image is identical to that of the calling process, except for a few parameters like the PID. The child gets a new PID.
- Exec: The forked child overwrites its own image with the code and data of the new program. This mechanism is called exec, and the child process is said to *exec* a new program, using one of the family of exec system calls. The PID and PPID of the exec'd process remain unchanged.
- Wait: The parent then executes the wait system call to *wait* for the child to complete. It picks up the exit status of the child and continues with its other functions. Note that a parent need not decide to wait for the child to terminate.

To get a better idea of this, let us explain with an example. When you enter ls to look at the contents of your current working directory, UNIX does a series of things to create an environment for ls and the run it:

- The shell has UNIX perform a fork. This creates a new process that the shell will use to run the ls program.
- The shell has UNIX perform an exec of the ls program. This replaces the shell program and data with the program and data for ls and then starts running that new program.
- The ls program is loaded into the new process context, replacing the text and data of the shell.
- The ls program performs its task, listing the contents of the current directory. In the meanwhile, the shell executes wait system call for ls to complete.

When a process is forked, the child has a different PID and PPID from its parent. However, it inherits most of the attributes of the parent. The important attributes that are inherited are:

- User name of the real and effective user (RUID and EUID): the owner of the process. The real owner is the user issuing the command, the effective user is the one determining access to system resources. RUID and EUID are usually the same, and the process has the same access rights the issuing user would have.
- Real and effective group owner (RGID and EGID): The real group owner of a process is the primary group of the user who started the process. The effective group owner is usually the same, except when SGID access mode has been applied to a file.
- The current directory from where the process was run.
- The file descriptors of all files opened by the parent process.
- Environment variables like HOME, PATH.

The inheritance here means that the child has its own copy of these parameters and thus can alter the environment it has inherited. But the modified environment is not available to the parent process.

How the Shell is created?

$$\begin{tikzcd} \textbf{init} \arrow[r, "fork"'] & \textbf{getty} \arrow[r, "fork-exec"'] & \textbf{login} \arrow[r, "fork-exec"'] & \textbf{shell} \end{tikzcd}$$

- When the system moves to multiuser mode, **init** forks and execs a **getty** for every active communication port.
- Each one of these **getty**'s prints the login prompt on the respective terminal and then goes off to sleep.
- When a user tries to log in, **getty** wakes up and fork-execs the **login** program to verify login name and password entered.
- On successful login, **login** for-execs the process representing the login shell.
- **init** goes off to sleep, waiting for the children to terminate. The processes **getty** and **login** overlay themselves.
- When the user logs out, it is intimated to **init**, which then wakes up and spawns another **getty** for that line to monitor the next login.

4. Internal and External Commands

From the process viewpoint, the shell recognizes three types of commands:

1. External commands: Commonly used commands like **cat**, **ls** etc. The shell creates a process for each of these commands while remaining their parent.
2. Shell scripts: The shell executes these scripts by spawning another shell, which then executes the commands listed in the script. The child shell becomes the parent of the commands that feature in the shell.
3. Internal commands: When an internal command is entered, it is directly executed by the shell. Similarly, variable assignment like x=5, doesn't generate a process either.

Note: Because the child process inherits the current working directory from its parent as one of the environmental parameters, it is necessary for the cd command not to spawn a child to achieve a change of directory. If this is allowed, after the child dies, control would revert to the parent and the original directory would be restored. Hence, cd is implemented as an internal command.

5. Process States and Zombies

At any instance of time, a process is in a particular state. A process after creation is in the *runnable* state. Once it starts running, it is in the *running* state. When a process requests for a resource (like disk I/O), it may have to wait. The process is said to be in *waiting* or *sleeping* state. A process can also be *suspended* by pressing a key (usually *Ctrl-z).*

When a process terminates, the kernel performs clean-up, assigns any children of the exiting process to be adopted by **init**, and sends the death of a child signal to the parent process, and converts the process into the zombie state.

A process in zombie state is not alive; it does not use any resources nor does any work. But it is not allowed to die until the exit is acknowledged by the parent process.

It is possible for the parent itself to die before the child dies. In such case, the child becomes an **orphan** and the kernel makes **init** the parent of the orphan. When this adopted child dies, **init** waits for its death.

6. Running Jobs in Background

The basic idea of a background job is simple. It's a program that can run without prompts or other manual interaction and can run in parallel with other active processes.

Interactive processes are initialized and controlled through a terminal session. In other words, there has to be someone connected to the system to start these processes; they are not started automatically as part of the system functions. These processes can run in the foreground, occupying the terminal that started the program, and you can't start other applications as long as this process is running in the foreground.

There are two ways of starting a job in the background – with the shell's & operator and the **nohup** command.

&: No Logging out

Ordinarily, when the shell runs a command for you, it waits until the command is completed. During this time, you cannot communicate with the shell. You can run a command that takes a long time to finish as a background job, so that you can be doing something else. To do this, use the & symbol at the end of the command line to direct the shell to execute the command in the background.

```
$ sort –o emp.dat emp.dat &
[1] 1413                  The job's PID
```

Note:

1. Observe that the shell acknowledges the background command with two numbers. First number [1] is the *job ID* of this command. The other number 1413 is the PID.
2. When you specify a command line in a pipeline to run in the background, all the commands are run in the background, not just the last command.
3. The shell remains the parent of the background process.

nohup: Log out Safely

A background job executed using & operator ceases to run when a user logs out. This is because, when you logout, the shell is killed and hence its children are also killed. The UNIX system provides nohup statement which when prefixed to a command, permits execution of the process even after the user has logged out. You must use the & with it as well.

The syntax for the nohup command is as follows:

```
nohup command-string [input-file] output-file &
```

If you try to run a command with nohup and haven't redirected the standard error, UNIX automatically places any error messages in a file named nohup.out in the directory from which the command was run.

In the following command, the sorted file and any error messages are placed in the file nohup.out.

```
$ nohup sort sales.dat &
1252
Sending output to nohup.out
```

Note that the shell has returned the PID (1252) of the process.
When the user logs out, the child turns into an orphan. The kernel handles such situations by reassigning the PPID of the orphan to the system's init process (PID 1) - the parent of all shells. When the user logs out, init takes over the parentage of any process run with nohup. In this way, you can kill a parent (the shell) without killing its child.

Additional Points
When you run a command in the background, the shell disconnects the standard input from the keyboard, but does not disconnect its standard output from the screen. So, output from the command, whenever it occurs, shows up on screen. It can be confusing if you are entering another command or using another program. Hence, make sure that both standard output and standard error are redirected suitably.

```
        $ find . –name "*.log" –print> log_file 2> err.dat &
OR      $ find . –name "*.log" –print> log_file 2> /dev/null &
```

Important:
1. You should relegate time-consuming or low-priority jobs to the background.
2. If you log out while a background job is running, it will be terminated.

7. nice: Job Execution with Low Priority
Processes in UNIX are sequentially assigned resources for execution. The kernel assigns the CPU to a process for a time slice; when the time elapses, the process is places in a queue. How the execution is scheduled depends on the priority assigned to the process.

The *nice* command is used to control background process dispatch priority.

The idea behind *nice* is that background jobs should demand less attention from the system than interactive processes.
Background jobs execute without a terminal attached and are usually run in the background for two reasons:
1. the job is expected to take a relatively long time to finish, and
2. the job's results are not needed immediately.

Interactive processes, however, are usually shells where the speed of execution is critical because it directly affects the system's apparent response time. It would therefore be nice

for everyone (others as well as you) to let interactive processes have priority over background work.
nice values are system dependent and typically range from 1 to 19.

A high *nice* value implies a lower priority. A program with a high nice number is friendly to other programs, other users and the system; it is not an important job. The lower the nice number, the more important a job is and the more resources it will take without sharing them.
Example:

```
        $ nice wc –l hugefile.txt
OR      $ nice wc –l hugefile.txt &
```

The default nice value is set to 10.
We can specify the nice value explicitly with –n *number* option where *number* is an offset to the default. If the –n *number* argument is present, the priority is incremented by that amount up to a limit of 20.
Example: `$ nice –n 5 wc –l hugefile.txt &`

8. Killing Processes with Signals

When you execute a command, one thing to keep in mind is that commands do not run in a vacuum. Many things can happen during a command execution that are not under the control of the command. The user of the command may press the interrupt key or send a kill command to the process, or the controlling terminal may become disconnected from the system. In UNIX, any of these events can cause a **signal** to be sent to the process. The default action when a process receives a signal is to terminate.

When a process ends normally, the program returns its *exit status* to the parent. This exit status is a number returned by the program providing the results of the program's execution.
Sometimes, you want or need to terminate a process.
The following are some reasons for stopping a process:

- It’s using too much CPU time.
- It’s running too long without producing the expected output.
- It’s producing too much output to the screen or to a disk file.
- It appears to have locked a terminal or some other session.
- It’s using the wrong files for input or output because of an operator or programming error.
- It’s no longer useful.

If the process to be stopped is a background process, use the kill command to get out of these situations. To stop a command that isn’t in the background, press <ctrl-c>.

To use kill, use either of these forms:
kill PID(s) OR kill –s NUMBER PID(s)
To kill a process whose PID is 123 use,

```
        $ kill 123
```

To kill several processes whose PIDs are 123, 342, and 73 use,

```
$ kill 123 342 73
```

Issuing the kill command sends a signal to a process. The default signal is SIGTERM signal (15). UNIX programs can send or receive more than 20 signals, each of which is represented by a number. (Use kill –l to list all signal names and numbers)

If the process ignores the signal SIGTERM, you can kill it with SIGKILL signal (9) as,

```
$ kill -9 123        OR     $ kill –s KILL 123
```

The system variable $! stores the PID of the last background job. You can kill the last background job without knowing its PID by specifying $ kill $!

Note: You can kill only those processes that you own; You can't kill processes of other users. To kill all background jobs, enter kill 0.

9. Job Control

A job is a name given to a group of processes that is typically created by piping a series of commands using pipeline character. You can use job control facilities to manipulate jobs. You can use job control facilities to,

1. Relegate a job to the background (bg)
2. Bring it back to the foreground (fg)
3. List the active jobs (jobs)
4. Suspend a foreground job (*[Ctrl-z]*)
5. Kill a job (kill)

The following examples demonstrate the different job control facilities.
Assume a process is taking a long time. You can suspend it by pressing *[Ctrl-z]*.

```
[1] + Suspended              wc –l hugefile.txt
```

A suspended job is not terminated. You can now relegate it to background by,

```
$ bg
```

You can start more jobs in the background any time:

```
$ sort employee.dat > sortedlist.dat &
[2]     530
$ grep 'director' emp.dat &
[3]     540
```

You can see a listing of these jobs using jobs command,

```
$ jobs
[3]  +  Running              grep 'director' emp.dat &
[2]  -  Running              sort employee.dat > sortedlist.dat &
[1]     Suspended     wc –l hugefile.txt
```

You can bring a job to foreground using fg %jobno OR fg %jobname as,

```
$ fg %2        OR     $ fg %sort
```

10. at And batch: Execute Later

UNIX provides facilities to schedule a job to run at a specified time of day. If the system load varies greatly throughout the day, it makes sense to schedule less important jobs at a time when the system load is low. The at and batch commands make such job scheduling possible.

at: One-Time Execution

To schedule one or more commands for a specified time, use the at command. With this command, you can specify a time, a date, or both.
For example,

```
$ at 14:23 Friday
at> lp /usr/sales/reports/*
at> echo “Files printed, Boss!” | mail -s”Job done” boss
[Ctrl-d]
commands will be executed using /usr/bin/bash
job 1041198880.a at Fri Oct 12 14:23:00 2007
```

The above job prints all files in the directory /usr/sales/reports and sends a user named boss some mail announcing that the print job was done.
All at jobs go into a queue known as at queue.at shows the job number, the date and time of scheduled execution. This job number is derived from the number of seconds elapsed since the Epoch. A user should remember this job number to control the job.

```
$ at 1 pm today
at> echo “^G^GLunch with Director at 1 PM^G^G” >
/dev/term/43
```

The above job will display the following message on your screen (/dev/term/43) at 1:00 PM, along with two beeps(^G^G).

Lunch with Director at 1 PM

To see which jobs you scheduled with at, enter at -l. Working with the preceding examples, you may see the following results:

job 756603300.a at Tue Sep 11 01:00:00 2007
job 756604200.a at Fri Sep 14 14:23:00 2007

The following forms show some of the keywords and operations permissible with at command:

at hh:mm	Schedules job at the hour (*hh*) and minute (*mm*) specified, using a 24-hour clock
at hh:mm month day year	Schedules job at the hour (*hh*), minute (*mm*), month, day, and year specified
at -l	Lists scheduled jobs
at now +count time-units	Schedules the job right now plus *count* number of *timeunits*; time units can be minutes, hours, days, or weeks
at –r job_id	Cancels the job with the job number matching *job_id*

batch: Execute in Batch Queue

The batch command lets the operating system decide an appropriate time to run a process. When you schedule a job with batch, UNIX starts and works on the process whenever the system load isn't too great.

To sort a collection of files, print the results, and notify the user named boss that the job is done, enter the following commands:

```
$ batch
sort /usr/sales/reports/* | lp
echo “Files printed, Boss!” | mailx -s”Job done” boss
```

The system returns the following response:

```
job 7789001234.b at Fri Sep 7 11:43:09 2007
```

The date and time listed are the date and time you pressed <Ctrl-d> to complete the batch command. When the job is complete, check your mail; anything that the commands normally display is mailed to you. Note that any job scheduled with batch command goes into a special at queue.

11. cron: Running jobs periodically

cron program is a daemon which is responsible for running repetitive tasks on a regular schedule. It is a perfect tool for running system administration tasks such as backup and system logfile maintenance. It can also be useful for ordinary users to schedule regular tasks including calendar reminders and report generation.

Both *at* and *batch* schedule commands on a one-time basis. To schedule commands or processes on a regular basis, you use the cron (short for *chronograph*) program. You specify the times and dates you want to run a command in crontab files. Times can be specified in terms of minutes, hours, days of the month, months of the year, or days of the week.

cron is listed in a shell script as one of the commands to run during a system boot-up sequence. Individual users don't have permission to run cron directly.

If there's nothing to do, cron “goes to sleep” and becomes inactive; it “wakes up” every minute, however, to see if there are commands to run.
cron looks for instructions to be performed in a control file in

```
/var/spool/cron/crontabs
```

After executing them, it goes back to sleep, only to wake up the next minute.

To a create a crontab file,

First use an editor to create a crontab file say cron.txt

Next use crontab command to place the file in the directory containing crontab files. crontab will create a file with filename same as user name and places it in /var/spool/cron/crontabs directory.

Alternately you can use crontab with –e option.

You can see the contents of your crontab file with crontab –l and remove them with crontab –r.

The cron system is managed by the cron daemon. It gets information about which programs and when they should run from the system's and users' crontab entries. The crontab files are stored in the file /var/spool/cron/crontabs/<user> where <user> is the login-id of the user. Only the root user has access to the system crontabs, while each user should only have access to his own crontabs.

A typical entry in crontab file

A typical entry in the crontab file of a user will have the following format.

```
        minute hour day-of-month month-of-year day-of-week command
```

where, Time-Field Options are as follows:

Field	Range
minute	00 through 59 Number of minutes after the hour
hour	00 through 23 (midnight is 00)
day-of-month	01 through 31
month-of-year	01 through 12
day-of-week	01 through 07 (Monday is 01, Sunday is 07)

The first five fields are time option fields. You must specify all five of these fields. Use an asterisk (*) in a field if you want to ignore that field.

Examples:

```
00-10 17 * 3.6.9.12 5 find / -newer .last_time –print >backuplist
```

In the above entry, the find command will be executed every minute in the first 10 minutes after 5 p.m. every Friday of the months March, June, September and December of every year.

```
30 07 * * 01 sort /usr/wwr/sales/weekly |mail -s”Weekly Sales” srm
```

In the above entry, the sort command will be executed with /usr/www/sales/weekly as argument and the output is mailed to a user named srm at 7:30 a.m. each Monday.

12. time: Timing Processes

The time command executes the specified command and displays the time usage on the terminal.

Example: You can find out the time taken to perform a sorting operation by preceding the sort command with time.

```
        $ time sort employee.dat > sortedlist.dat
        real    0m29.811s
        user    0m1.370s
        sys     0m9.990s
```

where,

the *real* time is the clock elapsed from the invocation of the command until its termination.
the *user* time shows the time spent by the program in executing itself.
the *sys* time indicates the time used by the kernel in doing work on behalf of a user process.
The sum of user time and sys time actually represents the CPU time. This could be significantly less than the real time on a heavily loaded system.

Conclusion

In this chapter, we saw an important abstraction of the UNIX operating system viz., processes. We also saw the mechanism of process creation, the attributes inherited by the child from the parent process as well as the shell's behavior when it encounters internal commands, external commands and shell scripts. This chapter also discussed background jobs, creation and controlling jobs as well as controlling processes using signals. We finally described three commands viz., at, batch and cron for process scheduling, with a discussion of time command for obtaining time usage statistics of process execution.